Performance-efficient Distributed Transfer and Transformation of Big Spatial Histopathology Datasets in the Cloud

Esma Yildirim

Department of Mathematics and Computer Science
Queensborough Community College of CUNY, Bayside, NY 11364

Abstract

Whole Slide Image (WSI) datasets are giga-pixel resolution, unstructured histopathology datasets that consist of extremely big files (each can be as large as multiple GBs in compressed format). These datasets have utility in a wide range of diagnostic and investigative pathology applications. However, the datasets present unique challenges: The size of the files, propriety data formats and lack of efficient parallel data access libraries limit the scalability of these applications.

Commercial clouds provide dynamic, cost-effective, scalable infrastructure to process these datasets, however we lack the tools and algorithms that will transfer/transform them onto the cloud seamlessly, providing faster speeds and scalable formats. In this study, I present novel algorithms that transfer these datasets onto the cloud while at the same time transforming them into symmetric scalable formats. My algorithms use intelligent file size distribution, and pipelining transfer and transformation tasks without introducing extra overhead to the underlying system. The algorithms, tested in the Amazon Web Services(AWS) cloud, outperform the widely used transfer tools and also outperform my previous work.

The transformed symmetric datasets are fed into two different analytics applications: a distributed implementation of a content-base image retrieval (CBIR) application for prostate carcinoma datasets and a deep convolutional neural network application for classification of breast cancer datasets. Although different in nature, both applications can easily work with my new symmetric data format and performance results show near-linear speed ups as the number of processors is increased.

Keywords: Big data applications, content-based image retrieval, cloud networks, distributed transfer algorithms, microscopy, whole slide images.

1. Introduction

Running biomedical analytics applications that process big WSI datasets in the cloud present unique challenges which can be classified as *dataset challenges*, *cloud architecture challenges* and *application challenges*. WSIs are very large tissue slide images in multi-giga-pixel resolution, produced by digital scanners and they have utility in a wide range of diagnostic and investigative pathology applications [1]. Considering a typical image can be as large as 200,000x200,000 pixels, each can occupy 30-50 GB space in uncompressed format which makes it unfeasible to bring into memory as a whole. Another problem with these datasets is that there is no uniform format and each digital scanner brand can produce images in propriety formats. There are only a handful of libraries that can read these images(e.g. Openslide [2], Bio-formats [3]) and they provide limited access capabilities for cloud storage systems (e.g. HDFS [4], AWS S3 [5]). The version 5 of Bio-formats can only work with parallel file systems(e.g. Lustre [6], GPFS [7]) and its pixel data format is not compatible with OpenCV [8] interfaces. The use of parallel processing algorithms and frameworks with whole slide image datasets is also very limited. Parallel access to the storage system is not considered as part of the parallelization process [9] [10]. They provide limited scalability results and store processed features rather than raw WSI pixel data [11]. In our previous work [1], we presented a distributed, dynamic asynchronous transfer and transformation algorithm that work on the Hadoop [12] ecosystem and this algorithm was able to scale seemlessly based on the capabilities of the underlying resources and store WSIs in raw format.

The second set of challenges can result from the architecture of the cloud. Although clouds can provide tremendous opportunities for a plethora of biomedical applications [13], [14], [15], [16], [17], [18], [19], blackbox object storage systems (e.g. AWS S3) and poor cloud networks are the main reasons analytics applications perform poorly [20]. These studies usually deal with smaller size image formats such as MRIs, microarrays, pCTs and do not come across with the problems presented by extremely large WSI datasets. The designed platforms also install their own storage system, thus have full control. In this case, they can easily predict the load of the system and have real-time access to the metadata and logs which are almost impossible with blackbox storage systems like AWS S3. The third challenge is usually application specific and also dependent on the data access patterns of the application. Teodoro et al. [9] presents that different access patterns of a plethora of image processing operations have great effect on the perfor-

⋆Email: eyildirim@qcc.cuny.edu

mance depending on the architecture used.

In this study, I target these challenges and present novel, distributed algorithms that can pipeline the transfer and transformation tasks of WSI datasets and scale seamlessly. The algorithms transform the unstructured format of raw WSIs to a symmetric binary format providing easy access to a large set of parallel and distributed analytics applications. Previous work could only provide processed feature formats [11], [21] which limit the range of applications that can work with them. The proposed algorithms outperform widely used cloud transfer tools such as *aws s3 cp* and *s3-dist-cp*, and our previous work [1] in an inter-/intra cloud data center setting using AWS S3 storage system and AWS EMR service.

I tested these symmetric binary datasets using two different case studies. The first study is a content-based image retrieval application [22] for searching cancerous patterns in a prostate carcinoma WSI dataset. The application is implemented on the Hadoop mapreduce framework using OpenCV library. The second study is a deep learning classification application that classifies regions as cancerous or not in a breast cancer WSI dataset. The application is implemented using Keras deep learning library on Tensorflow. The performance results for both applications provided near-linear speedups as the number of vcpus was increased.

Overall, the contribution of this study includes:

- Two novel, distributed data transfer and transformation algorithms/tools for converting WSI datasets into symmetric formats in the cloud.
- Access libraries to transformed datasets for implementing highly scalable analytics applications on Hadoop framework and Tensorflow.
- A distributed implementation of a Content-based Image Retrieval application with Hadoop MapReduce using prostate carcinoma WSI datasets
- A distributed implementation of Deep Neural Network application for classification of images as tumor and normal with Tensorflow/Keras using breast cancer WSI datasets.

In the following subsections, I explain the system and algorithm design (Section 2), case study applications (Section 3), give an extensive experimental study (Section 4) and finally end with conclusions and future work (Section 5).

2. System and Algorithm Design

In this section, the architecture of the underlying system is presented and three different algorithms are explained in detail: *Dynamic Asynchronous Scheduler* [1], *Greedy Scheduler* and *Pipelined Greedy Scheduler*.

2.1. Utilization of Native YARN applications on Hadoop Ecosystem

The algorithms are implemented as a native YARN application. YARN is the scheduler of the Hadoop ecosystem. Figure 1 presents the interactions between the Application Master task and transfer/transformation tasks. Bioinformatics library (e.g. Openslide) does not know how to interact with a distributed file system(e.g.HDFS) or an object storage system(e.g.S3). Therefore the transformation task where the WSI files are converted to symmetric binary files happen in the local disk of the transformation nodes. Most of the algorithm logic is implemented to the *Application Master*. The list of URLs of the WSI datasets is given to the *Application Master* in the form of a text file. The *Application Master* distributes these URLs to the transformation nodes each of which communicates with the source storage system to transfer the files into their local disks and starts the transformation process. Once the files are transformed they are transferred back to the destination storage system. Usually the destination storage system resides in the same region as the processing cluster. The transformation nodes can recognize different types of source and destination storage systems such as web servers, HDFS and S3 and uses the proper protocol to transfer these files from/to the storage system.

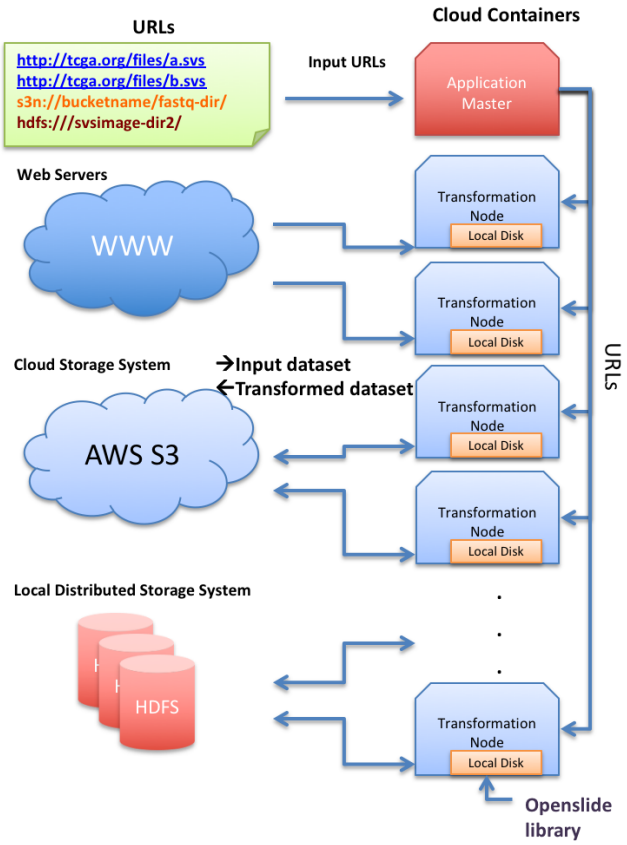

Figure 1: System architecture

2.2. Distributed Transfer/Transformation Algorithms

Three algorithms are presented in this section. *Dynamic Asynchronous Scheduler* is the algorithm developed in our previous work [1] and used as a baseline algorithm here. *Greedy Scheduler* and *Pipelined Greedy Scheduler* algorithms are novel

algorithms which use intelligent file distribution strategy and pipelining.

2.2.1. Dynamic Asynchronous Scheduler

This baseline algorithm assigns one transfer/transformation task to each transformation container. The containers are launched by YARN scheduler and as soon as one container finishes with its job another is launched. The number of parallel containers depends on the capabilities of the computer node and Hadoop configuration parameters. Outline of the algorithm is given in Algorithm 1.

The algorithm takes the input text file which contains the URLs for the WSI images, the resolution level and tile size as input parameters. WSI images can have multiple resolution levels. The *Application Master* container goes over the URLs in the text file and recognizes URL types. Then by using the appropriate protocols for each URL type, it interrogates the storage systems to make a global list of image files (Line 2). For each file URL in the list, the application master launches transformation containers who are responsible with the transfer, transformation and storage of the files on the local HDFS system or back on AWS S3 storage system (Lines 3-5). A simple scheduling algorithm is applied in which each container is assigned a single URL to process.

The application running on the transformation container recognizes the URL type and uses appropriate transfer protocol or API to transfer the file into its local disk (Lines 7-8). It then uses the bioinformatic library (Openslide or bio-format) to read the metadata of the file and gathers a list of coordinates based on the resolution level and tile size parameters(Lines 9-10). For each tile coordinate, it creates a key which consists of the WSI file name, x and y coordinates and size of the tile. The metadata about each tile is stored in the key because in this way, it is easier to implement specific Hadoop *partitioner* and *groupby* classes that can also be used to implement select-where clauses or join objects. This property allows various analytics applications and different access patterns to be possible with sequence files.

The transformation container application then reads the raw partial pixel data in the form of a tile, converts it into a value object and writes the key-value pair into HDFS or S3 as part of a binary sequence file (Lines 11-15). Sequence files are compressed binary object files which can be split by programming paradigms like Hadoop MapReduce [12] and Spark automatically.

2.2.2. Greedy Scheduler

Dynamic Asynchronous Scheduler creates a separate YARN container for each file to be transferred. However, launching/tearing down a container is a costly operation and as the number of files in a dataset increases, this cost increases respectively. The idea behind *Greedy Scheduler* algorithm is to minimize this cost by launching as many parallel containers as the underlying system allows. For example if the computer cluster has 12 processors, the algorithm then launches only 12 containers. In this case, the distribution of files among the containers becomes very important as we want all the containers to finish their tasks approximately at the same time. Therefore, *Greedy Scheduler* applies a greedy approach to load balance the distribution of files to the transformation containers. The outline of the algorithm is given in Algorithm 2.

After the list of URLs are constructed (Line 2), the files are sorted in descending order of their sizes (Line3). Then, the transformation container with the list of files of which total size is minimum is found and in every iteration of the loop the file in line is assigned to the list of that container (Lines 4-7). In doing so, I assign the largest files first to the minimum size list hence maintaining load balance. Then, a transformation container is launched for each list (Lines 8-10). The only difference in the algorithm of Transformation Containers is that now they have a list of files to transfer/transform instead of a single file (Lines 12-22).

2.2.3. Pipelined Greedy Scheduler

Once the file lists are assigned to the containers there is no reason that a transfer task of a file should wait the transformation task of the previous file in the list. Therefore, in *Pipelined Greedy Scheduler* algorithm, the transformation containers have a multi-threaded producer/consumer architecture. Each container task launches one transfer and one or more transformation threads where the threads share a common URL buffer. Once the transfer task finishes the local URL path is inserted to the buffer(Lines 14-18). The transformation task picks up a URL from the buffer and starts the transformation(Lines 20-28). In this case, the transfer and transformation tasks are pipelined. The outline of the algorithm is given in Algorithm 3.

The only difference in the *Application Master* compared to *Greedy Scheduler* is that once the file lists are constructed for the containers, they are sorted in ascending order of their file sizes(Line 9) . The reason for this is that it has been observed that the transformation takes more time than transfer. Therefore, the sooner the transformation task starts, the better for the pipelining overlap. As a result, the algorithm transfers smaller files first.

3. Case Studies

The transformed symmetric datasets can be fed into a variety of big data analysis frameworks such as Hadoop, Spark and Tensorflow, increasing their scalability levels. In this section, I present two case studies: A CBIR application that searches for cancerous patterns in a prostate carcinoma dataset from the Cancer Genome Atlas database and a deep learning classification application that classifies breast cancer images from the Camelyon Challenge dataset.

3.1. Case 1: Distributed Content-based Image Retrieval

If the transfer and transformation of the WSI dataset is the first step of a bioinformatics application pipeline, then the second step is the implementation of a scalable CBIR application that can access these datasets. I implemented a distributed content-based image retrieval application for prostate carcinoma

Algorithm 1 Dynamic_Asynchronous_Scheduler

Require: P_{WSI} : *Input path of text file containing WSI URLs* $\vee$ T_w : *Tile width* $\vee$ T_h : *Tile height* $\vee$ R : *Resolution level*

Application Master :
$L_{URL} \leftarrow$ **list_of_file_URLs**(P_{WSI})
for all URL_i *in* L_{URL} **do**
  launch_transformation_container(URL_i, R, T_w, T_h)
end for
Transformation Container :
$URL\_type\_x \leftarrow$ **recognize_URL_type**(URL_i)
$local\_file\_path \leftarrow$ **transfer_to_local_disk**(URL_type_x, URL_i)
$Width, Height \leftarrow$ **retrieve_size_of_WSI**($local\_file\_path$)
$List_{tiles} \leftarrow$ **retrieve_tile_coords**($Width, Height, local\_file\_path$)
for all $Tile_i$ *in* $List_{tiles}$ **do**
  $Key \leftarrow file\_name, x\_coord_i, y\_coord_i, tile\_width, tile\_height$
  $Value \leftarrow$ **read_tile_pixels_from_WSI**($local\_file\_path, x\_coord_i, y\_coord_i$)
  write_to_sequence_file($Key, Value, destination\_path$)
end for

Algorithm 2 Greedy_Scheduler

Require: P_{WSI} : *Input path of text file containing WSI URLs* $\vee$ T_w : *Tile width* $\vee$ T_h : *Tile height* $\vee$ R : *Resolution level* $\vee$ N : *Number of transformation containers*

Application Master :
$L_{URL} \leftarrow$ **list_of_file_URLs**(P_{WSI})
sort_descending_filesize(L_{URL})
for all URL_i *in* L_{URL} **do**
  $L_{URL_j} \leftarrow List\ of\ URLs\ of\ container_j$ *with minimum total file size*
  $L_{URL_j}.add(URL_i)$
end for
for all L_{URL_j} **do**
  launch_transformation_container(URL_i, R, T_w, T_h)
end for
$Transformation\ Container_j$:
for all URL_i *in* L_{URL_j} **do**
  $URL\_type\_x \leftarrow$ **recognize_URL_type**(URL_i)
  $local\_file\_path \leftarrow$ **transfer_to_local_disk**(URL_type_x, URL_i)
  $Width, Height \leftarrow$ **retrieve_size_of_WSI**($local\_file\_path$)
  $List_{tiles} \leftarrow$ **retrieve_tile_coords**($Width, Height, local\_file\_path$)
  for all $Tile_i$ *in* $List_{tiles}$ **do**
    $Key \leftarrow file\_name, x\_coord_i, y\_coord_i, tile\_width, tile\_height$
    $Value \leftarrow$ **read_tile_pixels_from_WSI**($local\_file\_path, x\_coord_i, y\_coord_i$)
    write_to_sequence_file($Key, Value, destination\_path$)
  end for
end for

WSI datasets from the Cancer Genome Atlas database using a coarse searching algorithm presented in [22].

In Figure 2, a query patch of glandular prostate carcinoma structure is searched in a tile cropped from a WSI image. In this case, a tile is actually a value object in our binary data format. A sliding window approach is used to search Region of Interests(ROIs) against the query patch. The size of the query patch defines the size of the sliding window. Both query patch and ROI are divided into rectangular circles and their color histograms are calculated using OpenCV library. Chi-square distance formula is used to calculate the distance between histograms, which provided the best results with our tests. After the distance of each rectangle is calculated, they are averaged. The mapper task reads the transformed binary files in the form of key-value pairs. The key of the map task is a user-defined key class that consists of file name, x and y coordinates of the tile, and width and height of the tile. The value is the BGR byte representation of the tile pixels. The query patch is provided to the map task using distributed cache feature of Hadoop ecosystem where the patch is shared among all tasks. The mapper applies the searching algorithm and calculates the distances. The output key is the distance and the output value is a user defined

Algorithm 3 Pipelined_Greedy_Scheduler

Require: P_{WSI} : *Input path of text file containing WSI URLs* $\vee$ T_w : *Tile width* $\vee$ T_h : *Tile height* $\vee$ R : *Resolution level* $\vee$ N : *Number of transformation containers*

Application Master :
$L_{URL} \leftarrow$ **list_of_file_URLs**(P_{WSI})
sort_descending_filesize(L_{URL})
for all URL_i *in* L_{URL} **do**
  $L_{URL_j} \leftarrow$ *List of URLs of container*$_j$ *with minimum total file size*
  $L_{URL_j}.add(URL_i)$
end for
for all L_{URL_j} **do**
  sort_ascending_filesize(L_{URL_j})
  launch_transformation_container(URL_i, R, T_w, T_h)
end for
Transformation Container$_j$:
Transfer Thread :
for all URL_i *in* L_{URL_j} **do**
  $URL\_type\_x \leftarrow$ **recognize_URL_type**(URL_i)
  $local\_file\_path \leftarrow$ **transfer_to_local_disk**(URL_type_x, URL_i)
  $prodcons\_buffer.add(local\_file\_path)$
end for
Transformation Thread/s :
for all $local\_file\_path_i$ *in* $prodcons\_buffer$ **do**
  $Width, Height \leftarrow$ **retrieve_size_of_WSI**$(local\_file\_path_i)$
  $List_{tiles} \leftarrow$ **retrieve_tile_coords**$(Width, Height, local\_file\_path_i)$
  for all $Tile_j$ *in* $List_{tiles}$ **do**
    $Key \leftarrow file\_name, x\_coord_j, y\_coord_j, tile\_width, tile\_height$
    $Value \leftarrow$ **read_tile_pixels_from_WSI**$(local\_file\_path_i, x\_coord_j, y\_coord_j)$
    write_to_sequence_file$(Key, Value, destination\_path)$
  end for
end for

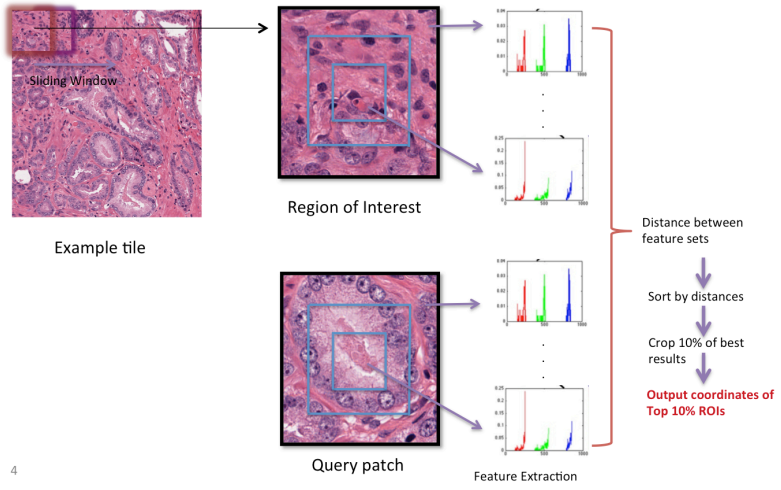

Figure 2: Content-based Image Retrieval Application - Coarse Searching Algorithm

class which consists of file name, x and y coordinates of the tile, width and height of the tile, x and y coordinates of the ROI, and width and height of the ROI. Making the distance the output key of the mapper tasks allows the reducer task to automatically sort the results based on the distance. Once the results are sorted we can pick the top best ROIs.

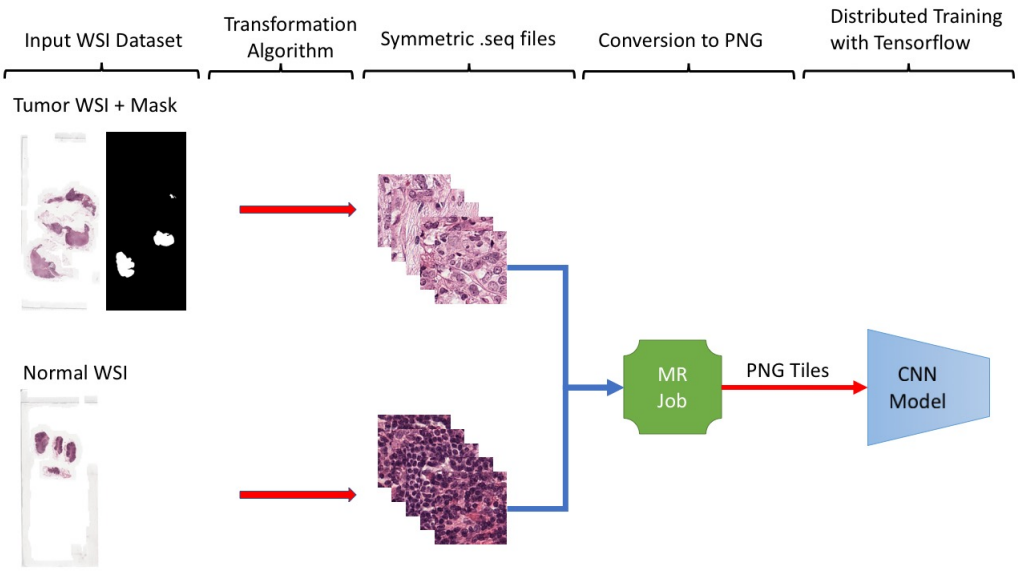

Figure 3: Classification Application Pipeline

3.2. Case 2: Distributed Classification via Deep Neural Networks

The second application is a deep convolutional neural network for classification of breast cancer images from the Camelyon Dataset. The dataset consists of tumor WSIs, tumor mask WSIs and normal WSIs (Figure 3). Our transfer/transformation algorithms can add an additional label (0 for normal, 1 for tumor) to the key as these datasets are converted into symmetric binary format. For tumor tiles, if the tile tumor mask is more than 50% white, I assign the label as *tumor*. For normal WSI images, a different thresholding method is applied. If the average of the R, G and B pixels is less than 150 and the percentage of such pixels in the tile is greater than 80%, I assign the label as *normal*. These values can be changed based on the background color of the WSI image.

Although Hadoop and Spark applications can access the transformed datasets directly because of their Java serialization compatibility, Tensorflow applications need an extra step. Therefore, before the training process starts, I add another MapReduce job to the pipeline to transfer the transformed datasets into the local filesystem in the form of PNG patches. Then, our distributed Convolutional Neural Network application starts training. I have not been able to find a TFRecordReader class that can read Java serialized binary sequence files. In our future work, I plan to bypass Java serialization and write our own serializer for Hadoop and a TFRecordReader class in C++ for Tensorflow so the incompatibility issue is solved.

The CancerNet model by Adrian Rosebrock [23] is used for my classification application. The model uses exclusively 3×3 CONV filters, stacks multiple 3×3 CONV filters on top of each other prior to performing max-pooling and uses depth-wise separable convolution rather than standard convolution layers. Their model used 48x48 patches from Kaggle Breast Histopathology dataset [24]. Since we can have any size patches through our transformation algorithms on the fly, I used a larger patch size 128x128 while creating the symmetric datasets. I also enabled distributed training by using Tensorflow's MultiWorkerMirroredStrategy. Therefore, I increased the batch size as well from 32 to $32 \times \#vcpus$ initially. The dataset was divided into training, validation and testing datasets. 80% of the dataset was used as the training set while the remaining %20 was used as the testing set. 10% of the 20% was also set aside as validation set. The application scaled well as I increased the number of vcpus and provided very high accuracy results (see Section 4.5).

4. Experiments

In this section, I present an extensive experimental study showing the performance and accuracy of our algorithms and case study applications. AWS S3 storage system and AWS EMR service is used for the experiments. 4 different storage locations are selected for the datasets: Virginia, Oregon, Canada and Frankfurt to test different bandwidth and RTTs. For EMR clusters, memory optimized instances are selected since these applications require large memory space. The node types I used range from 4vcpu (m5.xlarge) to 32 vcpu (m5.8xlarge) instances. However for the classification application only c3 type instances were available in AWS EMR with Tensorflow deployment. Therefore, I used c3.2xlarge, c3.4xlarge and c3.8xlarge instances in the experiments.

4.1. Comparison of Total Transfer Time

In this experiment, the transfer time of the dataset from the source S3 storage system to the local file system of the EMR

cluster is measured. The transformation feature of the algorithms were disabled just to see the total transfer time. I compared Dynamic Asynchronous Scheduler (DAS) and Greedy Scheduler (GS) algorithms against widely used transfer tools in the cloud: *aws s3 cp* and *s3-dist-cp*. The reason why I only measured the transfer time is that these tools are not capable of transforming the datasets. *aws s3 cp* is the default transfer tool of the AWS client interface and is a multi-threaded application. *s3-dist-cp* is a Mapreduce application that distributes the transfer of files into reducer tasks.

In Figure 4, I transferred a 100 WSI dataset(approximately 19GB) using m5.xlarge instance clusters ranging the vcpu number between 4-32. The results show similar characteristics for all source regions regardless of their different RTTs. Although *aws s3 cp* outperforms the rest of the tools/algorithms for 4 vcpu results, it cannot scale beyond the limits of a single instance due to its shared memory architecture. On the other hand, a multi-threaded architecture has less overhead compared to a multi-container architecture. I believe its superior performance at 4 vcpus is because of its threaded implementation providing less overhead.

As I mentioned in Section 2, GS algorithm is optimized compared to DAS algorithm because it uses less number of containers, this statement also applies to *s3-dist-cp* as it uses as many number of reducer containers as the underlying Hadoop configuration allows. In all of the cases, GS algorithm outperforms *s3-dist-cp* and when the parallelism level is high DAS algorithm slightly outperforms *s3-dist-cp* as well.

When we look at the total transfer time in different regions Canada has the best performance although its RTT ($\approx 15ms$) is higher than Virginia(less than $\approx 1ms$). I believe, the intra network traffic of Virginia datacenter is much higher than the inter network traffic between Canada and Virginia. This indicates that RTT is a less effective factor in transfer time compared to the network traffic. Similarly, the total time of Oregon-Virginia (RTT $\approx 50ms$) transfers is very close to Frankfurt-Virginia (RTT $\approx 80ms$) transfers proving that traffic is a much more definitive factor than RTT when it comes to cloud networks.

PPGS algorithm is not included in the results because it only differs from GS when transformation is enabled.

4.2. Comparison of Total Transfer/Transformation Time

In the second experiment, I transferred a 50 WSI dataset (approximately 15GB in compressed format). The replicas of the dataset are again placed in Virginia, Oregon, Frankfurt and Canada data centers. An EMR cluster of m5.4xlarge nodes is created in Virginia data center and the transfer/transformation algorithms were tested on this cluster. The destination is the S3 system of Virginia data center.

Figure 5 presents the total time it takes to convert WSI files into symmetric binary format as I increase the number of vcpus in the cluster between the range [8-32]. In the figure, DAS stands for *Dynamic Asynchronous Scheduler*, GS stands for *Greedy Scheduler* and PPGS stands for *Pipelined Greedy Scheduler*. As the number of vcpus is increased, the number of transformation containers launched increases for GS and PPGS algorithms as well. As a result, the number of files per container dramatically decreases, hence the optimizations I do in these algorithms lose their effect. That's why for 32 vcpu case, all the algorithms perform similarly. For all the other cases, GS performs better than DAS and PPGS performs better than GS algorithm. Also, as I increase the number of vcpus, the total transfer/transformation time decreases for all cases. The same observation that there was not much difference between the total times for different data centers were made in total transfer/transformation times as well. Then again, the highest total time value belongs to Frankfurt-Virginia transfers as they are transoceanic transfers. On the other hand, Virginia-Virginia and Canada-Virginia transfers almost took the same amount of time. Based on these results, it is clear that my new algorithms (GS and PPGS) outperform the baseline algorithm (DAS) and they scale well with the number of vcpus.

4.3. Effect of Pipelining Transfer and Transformation Tasks

Although PPGS algorithm outperforms both GS and DAS algorithms, I expected its performance to be better. To analyze the results, I took a closer look into three sample transfers between Canada - Virginia data centers on an 8 vcpu setting.

In Figure 6, I present a time series graphic for DAS algorithm job execution times. Since I did not have any control over which job/task was assigned to which cpu, the figure presents each file transfer separately thorough their job ids. It is clear that initially 8 jobs are started about the same time and as soon as one finishes another one is launched. In total 50 containers(corresponding to jobs) were launched.

In Figure 7, I present a time series graphic for GS algorithm job execution times. 8 jobs were run each one transfer/transforming a batch of files. By looking at the figure, we can see that the total execution times are quite load-balanced.

In Figure 8, I present a time series graphic for PPGS job execution times. Since each job is a 2-thread application the y-axis shows the job id and the thread type. For example, 1_xfer refers to job 1 - transfer thread while 1_xform refers to job 1 - transform thread. Between the transfer and transformation threads of a job, same colors represent the same file transfer and transformation respectively. The first interesting observation was that the transfer and transformation threads overlapped very well. The question is why is total run time is NOT much better compared to GS? The second observation was that the only times a transformation thread waited on the transfer thread were the white breaks in the timelines and there were very few of them. So the lack of performance was due to the transformation threads waiting each other rather than waiting for transfer threads to finish.

With these observations in mind, I decided to do more extensive tests parametrizing the number of parallel transformation threads so that I can create more than one transformation thread per transfer thread.

Figure 9 presents the effect of changing the ratio of #transfer threads to #transformation threads of PPGS algorithm on the total time. I set # job containers to 3 in Figure 9.a and use

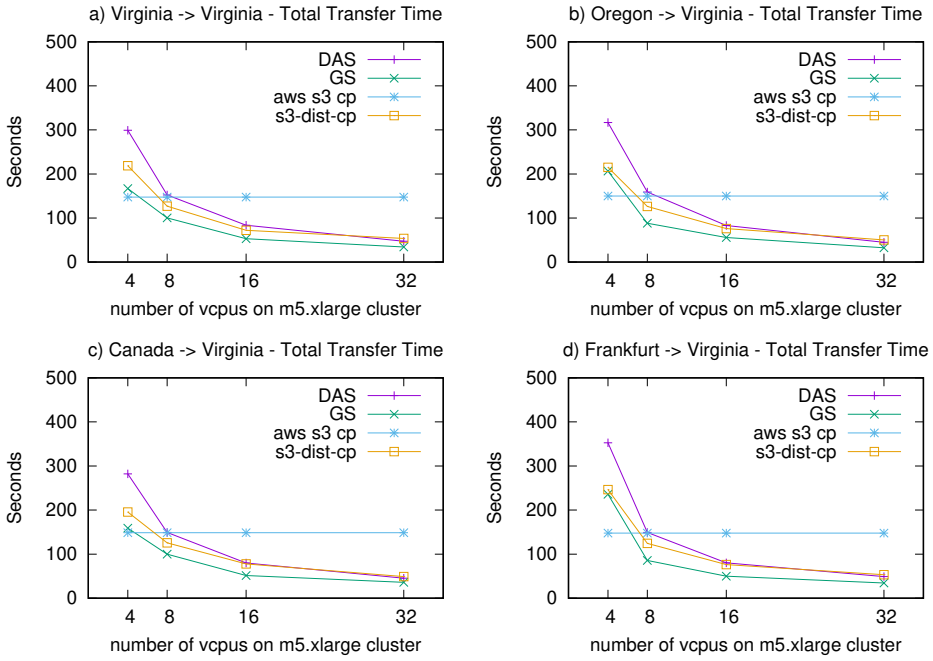

Figure 4: Performance Comparison of Total Transfer Time

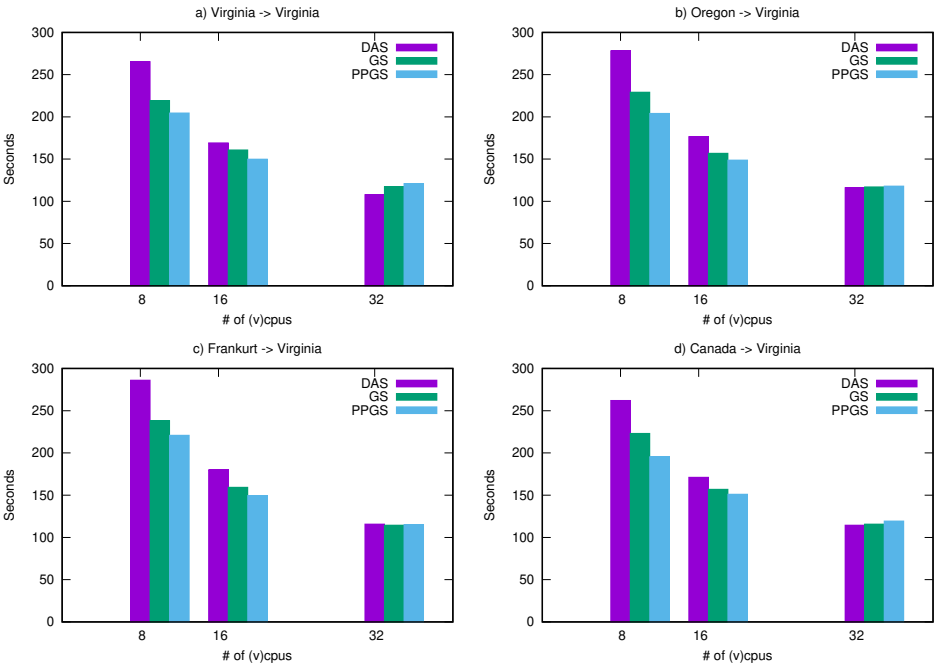

Figure 5: Performance Comparison of Transfer/Transformation Algorithms on AWS S3 and EMR

different instance types, the smallest being m5.xlarge having 4 vpus. The YARN scheduler allocates one container to the application master, therefore for true parallelism, I set the # containers to 3. Similarly, for Figure 9.b parallelism is set to 7 for m5.2xlarge instance (8vpus) and to 15 for m5.4xlarge instance (16) in Figure 9.c. The first observation made is that as I increase the # of transformation threads total time decreases but eventually becomes stable. The slope of the decrease is higher for more powerful instance types (instances with more vcpus). For example, in Figure 9.a as I increase the # of transformation threads for m5.4xlarge instance the total time quickly flattens although it improves for PPGS compared to DAS and GS. I looked into CPU utilization and saw that it was maxed out at 1/2 ratio. I increase the ratio only once after this point to prove the flattening of the curve. As the instance type changes, more cpu time becomes available thus PPGS algorithm performs bet-

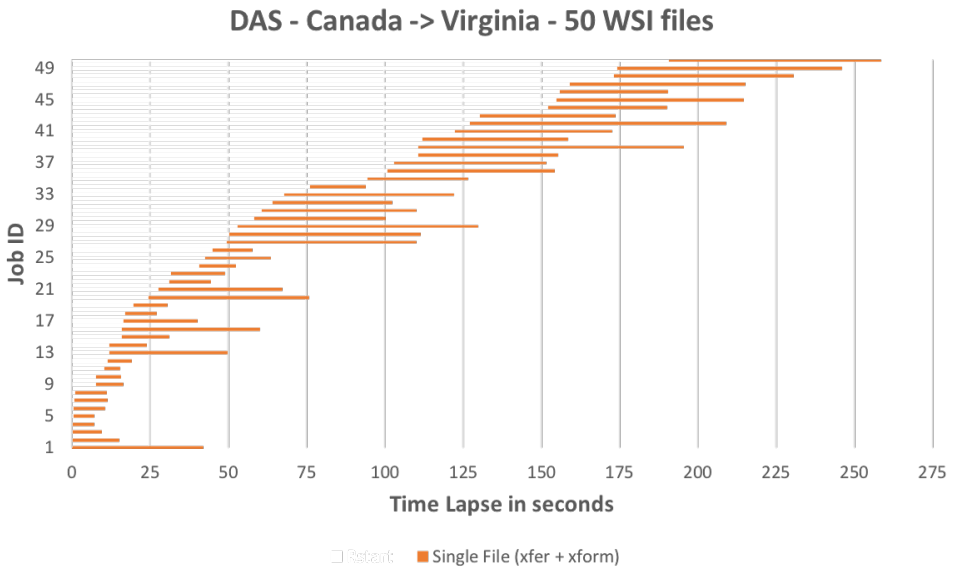

Figure 6: DAS

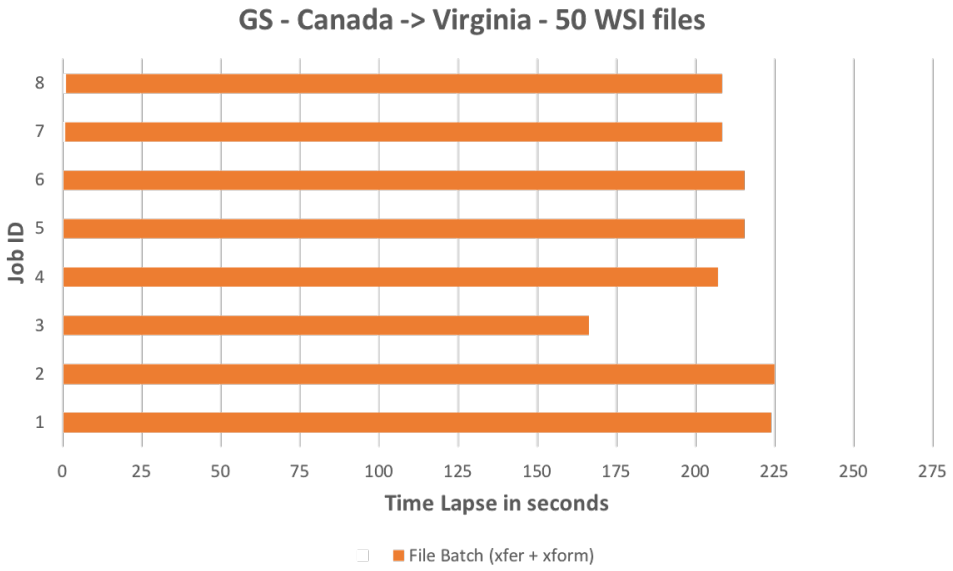

Figure 7: GS

ter. Most of the time, the time flattening occurred due to maxed out CPU utilization except for the case of reaching the network bandwidth limit. This happened only in two cases: first, when #containers = 7 and PPGS-1/3 and then second, when # containers = 15 and PPGS-1/1.

Based on these findings, I decided to design our transfer/transformation tool to parametrize the #containers and #transformation threads to be set by the user considering different instance types and their processing powers. In our future work, I intend to automatically set optimal values for these parameters.

4.4. Accuracy and Performance Results of CBIR Application

In this experiment, I tested the CBIR application on a 25-split transformed dataset. A split stands for a block of symmetric data for which a separate mapper/reducer task is created. A 100x100 pixel query image which contains a single gland and a 300x300 pixel query image which contains clusters of glandular structures were used (Figure 10 and Figure 11). The application takes two input parameters: *Kbins*, which stands for number of rectangular circles the ROIs are divided and *Olap*, which stands for the overlap percentage between the sliding windows. 4 different parameter combinations were used in the tests for *Kbins* and *Olap* respectively: 5-0.25, 5-0.5, 10-0.25 and 10-0.5. Figure 10 presents the accuracy results of the

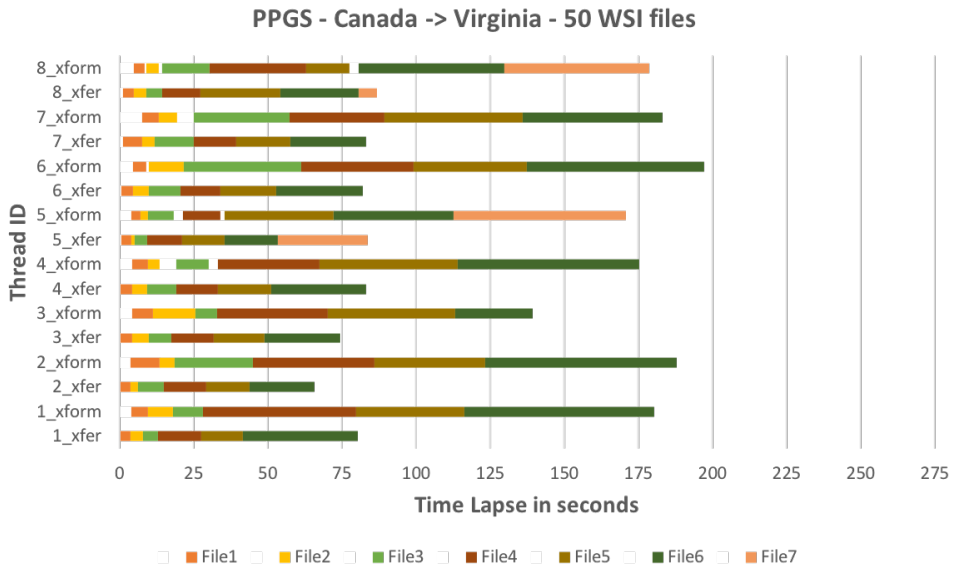

Figure 8: PPGS

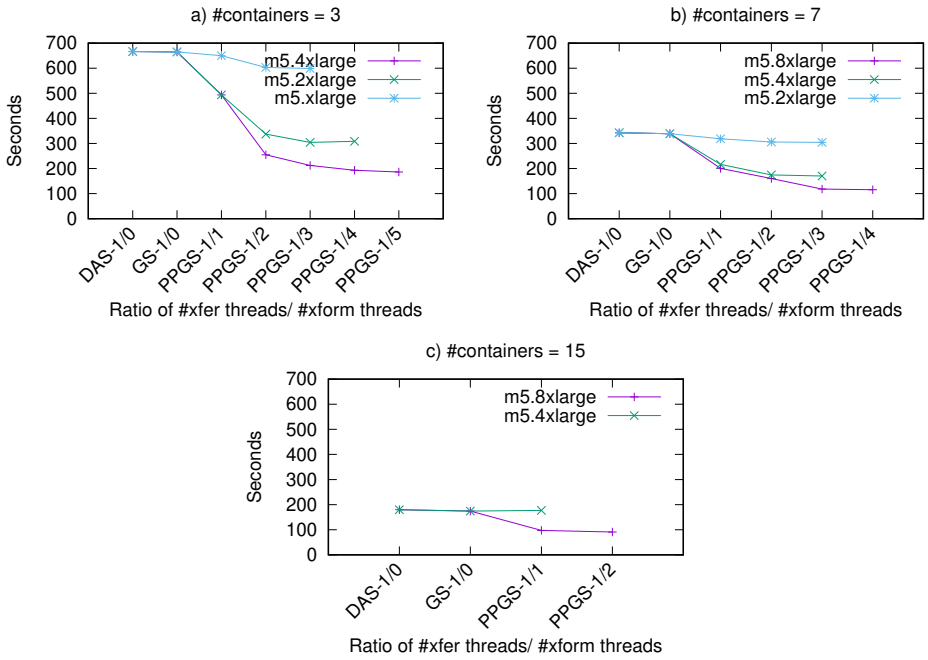

Figure 9: Effect of ratio of #transfer threads / #transformation threads

top ten ROIs for the specific 100x100 pixel query patch based on different parameter settings. In almost all of the cases, similar glandular structures were found.These structures were more similar to the query patch for the top 5 results. It was interesting to see that *Olap* parameter was more effective on the results than *Kbins* parameter. The results (Figure 11) were better for 300x300 query image where a cluster of glandular structures were searched in the dataset. In all of the cases, the exact query image was found as the top similar result. All the cases presented very similar clustered glandular structure images. Again *Olap* parameter was more effective compared to *Kbins*.

Figure 12 presents the performance results of running CBIR application on an EMR cluster of m5.8xlarge and m5.4xlarge instances as I increase the number of nodes parameter. The CBIR turned out to be a memory-intensive application. For 100x100 query patch the memory of an m5.8xlarge instance was sufficient to process a split of the dataset without giving errors. For 300x300 query patch, the memory of a m5.4xlarge instance was sufficient. As I increased the number of instances for both cases between the range [1-8], the total execution time decreased dramatically (Figure 12.a and c). Increasing *Kbins* from 5 to 10 also increased the execution time as well as in-

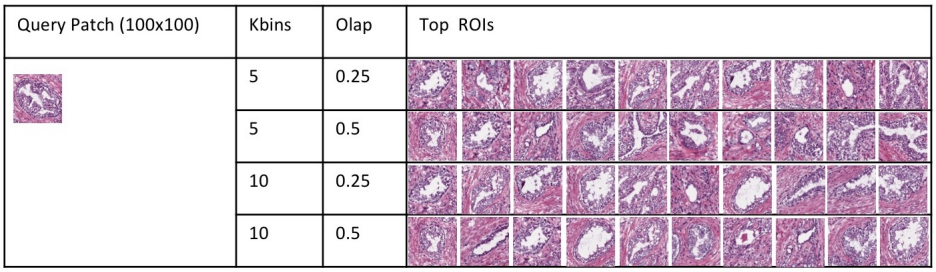

Figure 10: Accuracy Results of 100x100 query patch - single gland

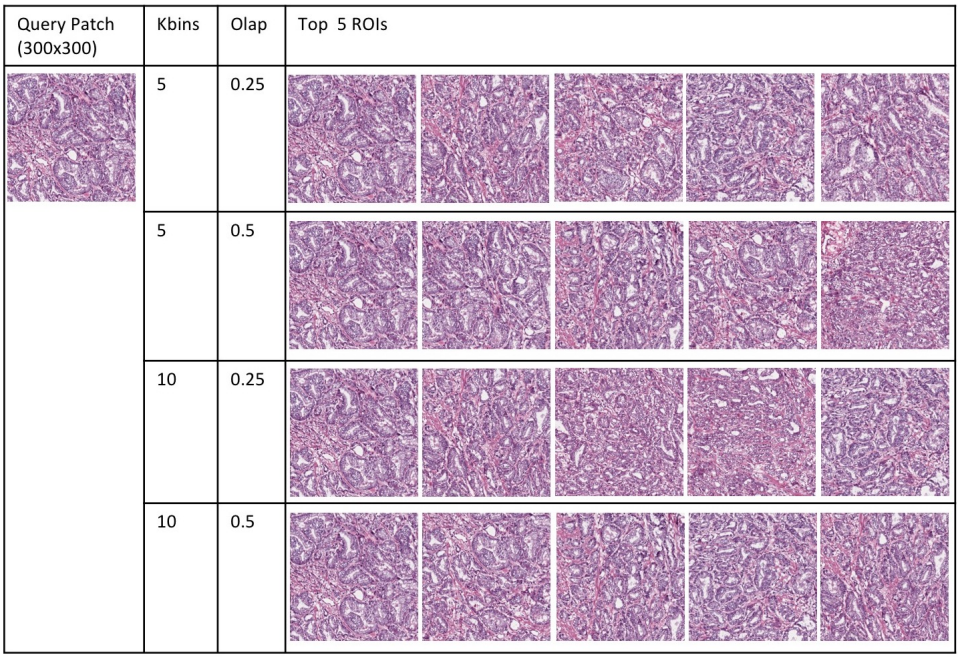

Figure 11: Accuracy Results of 300x300 query patch - clusters of glands

creasing *Olap* from 0.25 to 0.5. It took more time to search for 100x100 query patch than 300x300 query patch. I also calculated the speed up values by dividing the total execution time with 1 instance to total execution time with n instances. Their corresponding speed up graphics (Figure 12.b and d) show almost linear increase as I increase the number of instances indicating that trend might go on for larger number of instances. These results prove that CBIR application works well with symmetric binary datasets in terms of accuracy and scalable execution time.

4.5. Accuracy and Performance Results of Classification Application

I tested the second application with a WSI dataset of 21(10 tumor + 11 normal) breast cancer images from the Camelyon Dataset. The transformed files consist of 61 symmetric splits. The initial batch size was set to $32 \times \#vcpus$. The #vcpus were ranged between 8-32 on c3 types of EMR instances. The #epochs were set to 20. Figure 13 shows the comparison of training accuracy and loss to validation accuracy and loss, as well as testing accuracy, sensitivity and specificity.

Initially, I increased the total batch size in proportion of the number of vcpus but observed that this approach only lengthened the time it took for the validation accuracy and loss to converge with the training accuracy and loss. Another interesting observation was that if I keep the batch size fixed but increase the number of vcpus, the optimal number of epochs remains the same (6 epochs).

I also got 97%-99% accuracy, sensitivity and specificity on the testing dataset results for different parallelism levels (Figure 13). These results are much higher compared to the original CancerNet results on Kaggle dataset (83% accuracy).

The performance results of the experiments are presented in Figure 14. Per epoch time decreased significantly as I increased the number of vpus (Figure 14.a). Increasing the batch size as I increase the number of vcpus did not affect the per epoch time, however it caused the optimal number of epochs for the validation accuracy and loss to converge into training accuracy and loss to increase. Therefore, I kept the batch size fixed and did not show the results of differentiating batch sizes in this paper.

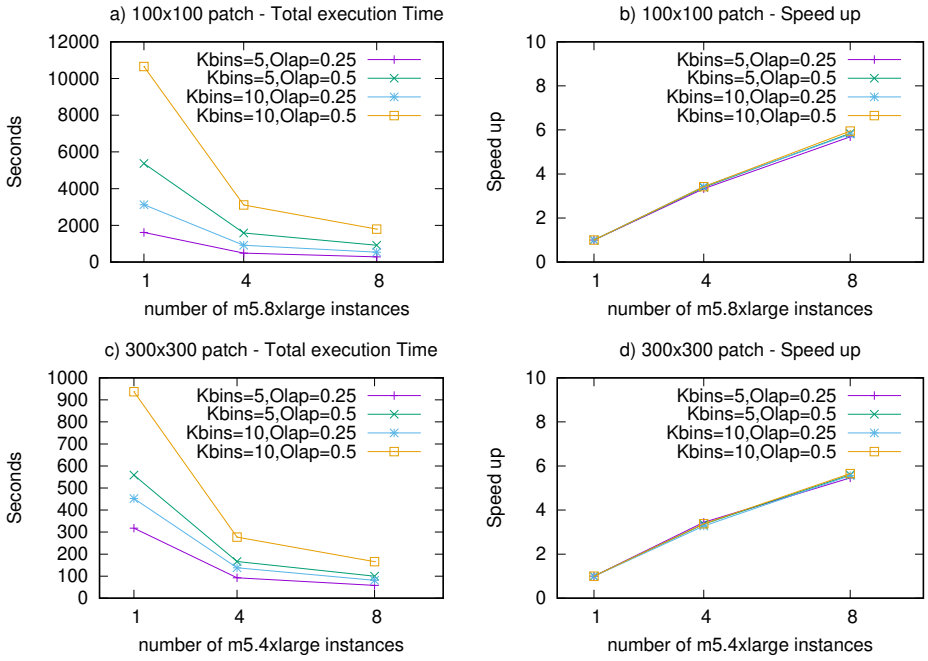

Figure 12: Performance Results of CBIR Application on AWS S3 and EMR

The conversion time of .seq files to png patches also decreased significantly as I increased the number of vpcus(Figure 14.b).

All of the cases took 6 epochs for the validation accuracy to reach to at least 95% accuracy. Therefore, I set this number as the optimal number of epochs and calculated the optimal training time by summing the per epoch time of the first 6 epochs. Although the time decrease slowed between 16 and 32, still there was significant decrease as I increased the vcpu number(Figure 14.c).

The last measure presented is the speed up, which is calculated as dividing the optimal training time of the base case (8vcpus) by the optimal training time of n vcpus(Figure 14.d). The results showed near linear increase as I increased the number of vcpus.

5. Conclusion and Future Work

In this study, I presented two novel distributed transfer/ transformation algorithms that used intelligent file distribution and pipelining as optimization techniques. The algorithms scaled well in an AWS cloud setting and outperformed the algorithm from our previous work and commonly used transfer algorithms in the cloud such as *aws s3 cp* and *s3-dist-cp*. The transformed datasets were tested with a distributed CBIR application for prostate cancer image datasets, which showed near linear speed up in terms of total execution time, as well as a distributed deep learning classification application of breast cancer image datasets, which also showed near linear speed ups for optimal training time. As future work, I intend to develop serialization methods for more flexible reader libraries for our transformation formats to target a large range of scalable analytics applications. I also plan to improve my transfer algorithms by setting the optimal parallelism levels automatically.

Acknowledgment

This work was supported by PSC CUNY Grant # 62177-00 50.

References

[1] E. Yildirim, D. J. Foran, Parallel versus distributed data access for gigapixel-resolution histology images: Challenges and opportunities, IEEE journal of biomedical and health informatics 21 (4) (2017) 1049–1057.

[2] A. Goode, B. Gilbert, J. Harkes, D. Jukic, M. Satyanarayanan, Openslide: A vendor-neutral software foundation for digital pathology, Journal of pathology informatics 4.

[3] J. Moore, M. Linkert, C. Blackburn, M. Carroll, R. K. Ferguson, H. Flynn, K. Gillen, R. Leigh, S. Li, D. Lindner, et al., Omero and bio-formats 5: flexible access to large bioimaging datasets at scale, in: Medical Imaging 2015: Image Processing, Vol. 9413, International Society for Optics and Photonics, 2015, p. 941307.

[4] D. Borthakur, et al., Hdfs architecture guide, Hadoop Apache Project 53 (1-13) (2008) 2.

[5] Amazon simple storage system (2021).
URL `https://aws.amazon.com/s3/`

[6] P. Braam, The lustre storage architecture, arXiv preprint arXiv:1903.01955.

[7] F. B. Schmuck, R. L. Haskin, Gpfs: A shared-disk file system for large computing clusters., in: FAST, Vol. 2, 2002.

[8] Opencv library (2021).
URL `https://opencv.org`

[9] G. Teodoro, T. Kurc, J. Kong, L. Cooper, J. Saltz, Comparative performance analysis of intel (r) xeon phi (tm), gpu, and cpu: a case study from microscopy image analysis, in: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IEEE, 2014, pp. 1063–1072.

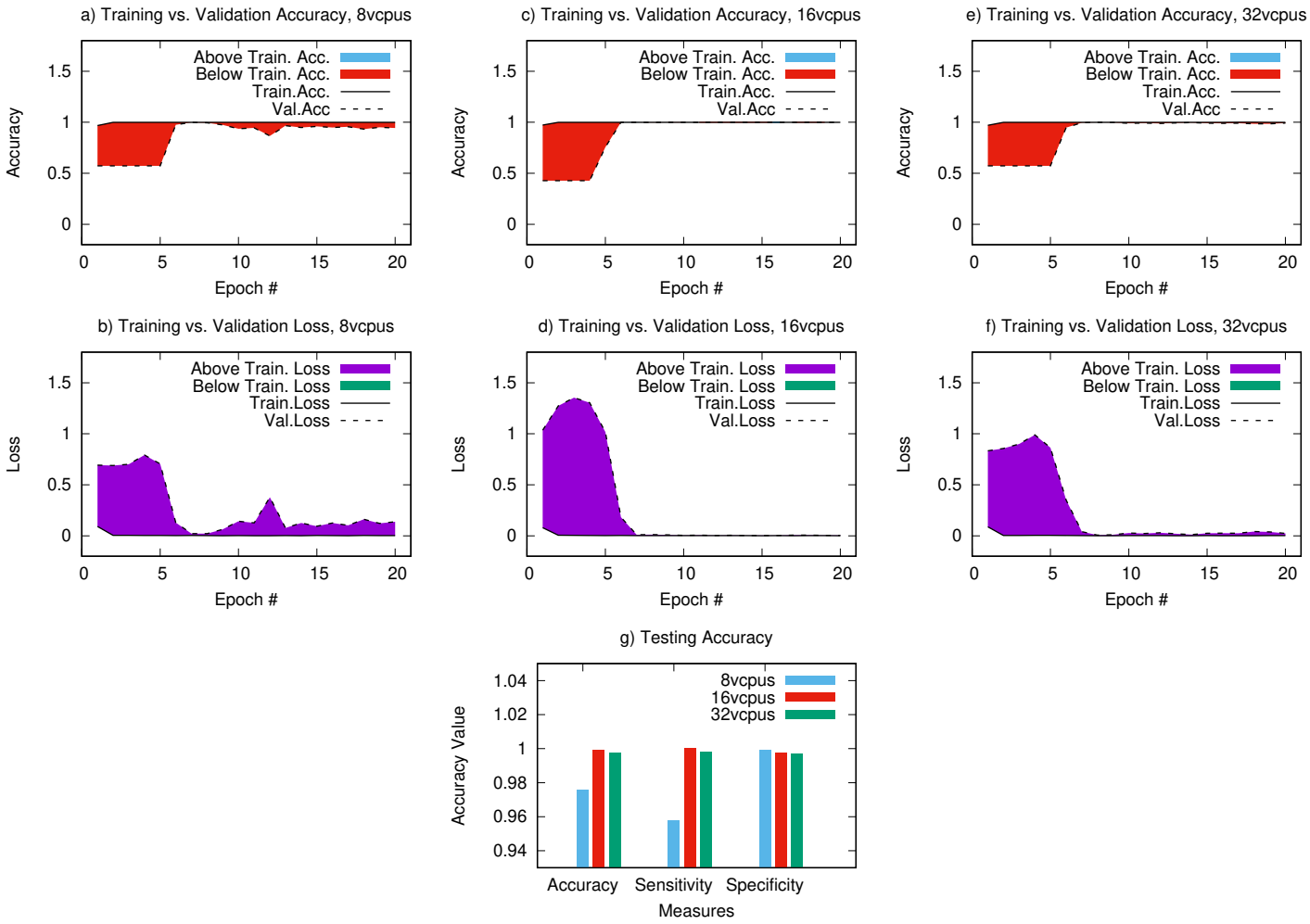

Figure 13: Accuracy Results of the Classification Application

[10] N. Zerbe, P. Hufnagl, K. Schlüns, Distributed computing in image analysis using open source frameworks and application to image sharpness assessment of histological whole slide images, in: Diagnostic pathology, Vol. 6, BioMed Central, 2011, p. S16.

[11] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, J. Saltz, Hadoop-gis: A high performance spatial data warehousing system over mapreduce, in: Proceedings of the VLDB Endowment International Conference on Very Large Data Bases, Vol. 6, NIH Public Access, 2013.

[12] Hadoop (2021).
URL `https://hadoop.apache.org`

[13] R. Chard, R. Madduri, N. T. Karonis, K. Chard, K. L. Duffin, C. E. Ordoñez, T. D. Uram, J. Fleischauer, I. T. Foster, M. E. Papka, et al., Scalable pct image reconstruction delivered as a cloud service, IEEE Transactions on Cloud Computing 6 (1) (2015) 182–195.

[14] L. Parsonson, S. Grimm, A. Bajwa, L. Bourn, L. Bai, A cloud computing medical image analysis and collaboration platform, in: International Conference on Cloud Computing and Services Science, Springer, 2011, pp. 207–224.

[15] G. C. Kagadis, C. Kloukinas, K. Moore, J. Philbin, P. Papadimitroulas, C. Alexakos, P. G. Nagy, D. Visvikis, W. R. Hendee, Cloud computing in medical imaging, Medical physics 40 (7).

[16] R. K. Madduri, D. Sulakhe, L. Lacinski, B. Liu, A. Rodriguez, K. Chard, U. J. Dave, I. T. Foster, Experiences building globus genomics: a next-generation sequencing analysis service using galaxy, globus, and amazon web services, Concurrency and Computation: Practice and Experience 26 (13) (2014) 2266–2279.

[17] F. Milletari, J. Frei, M. Aboulatta, G. Vivar, S.-A. Ahmadi, Cloud deployment of high-resolution medical image analysis with tomaat, IEEE journal of biomedical and health informatics 23 (3) (2018) 969–977.

[18] T. M. Godinho, C. Viana-Ferreira, L. A. B. Silva, C. Costa, A routing mechanism for cloud outsourcing of medical imaging repositories, IEEE journal of biomedical and health informatics 20 (1) (2014) 367–375.

[19] B. S. Harvey, S.-Y. Ji, Cloud-scale genomic signals processing for robust large-scale cancer genomic microarray data analysis, IEEE journal of biomedical and health informatics 21 (1) (2015) 238–245.

[20] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, N. J. Wright, Performance analysis of high performance computing applications on the amazon web services cloud, in: 2010 IEEE second international conference on cloud computing technology and science, IEEE, 2010, pp. 159–168.

[21] E. Bremer, J. Almeida, J. Saltz, Representing whole slide cancer image features with hilbert curves, arXiv preprint arXiv:2005.06469.

[22] X. Qi, D. Wang, I. Rodero, J. Diaz-Montes, R. H. Gensure, F. Xing, H. Zhong, L. Goodell, M. Parashar, D. J. Foran, et al., Content-based histopathology image retrieval using cometcloud, BMC bioinformatics 15 (1) (2014) 287.

[23] Breast cancer classification with keras and deep learning (2021).
URL `https://www.pyimagesearch.com/2019/02/18/breast-cancer-classification-with-keras-and-deep-learning/`

[24] Breast histopathology images (2021).
URL `https://www.kaggle.com/paultimothymooney/breast-histopathology-images`

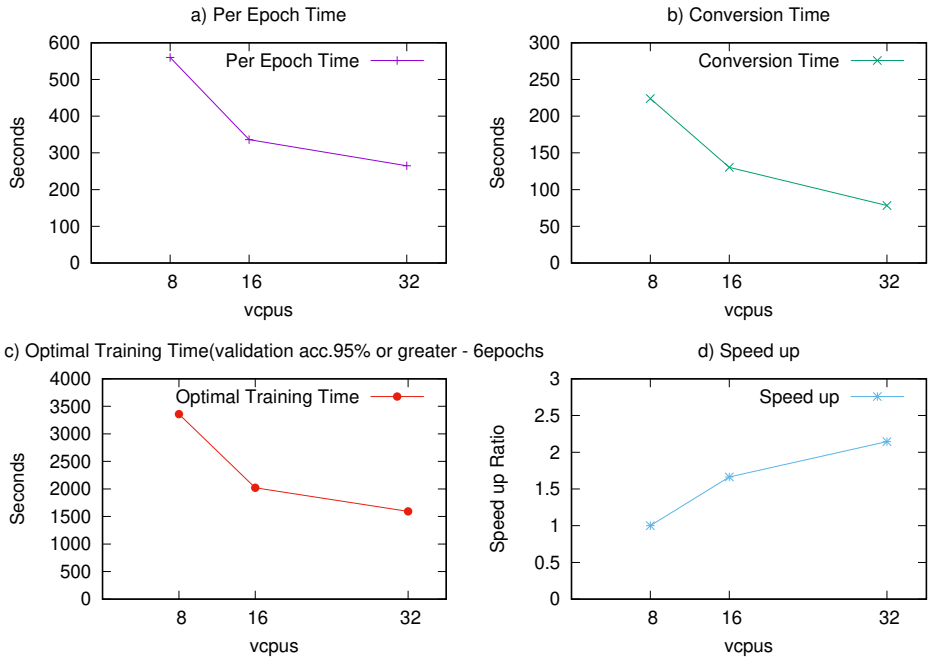

Figure 14: Performance Results of the Classification Application